ARPA ORDER NO.: 3460/3681
0P10 Information Processing
Techniques

R-2411-ARPA
February 1980

EP-2: An Exemplary Programming System

W. S. Faught, D. A. Waterman, P. Klahr, S. J. Rosenschein,
D. M. Gorlin, S. J. Tepper

A Report prepared for

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under Contract No. MDA903-78-C-0029.

Library of Congress Cataloging in Publication Data
Main entry under title:

EP-2, an exemplary programming system.

([Report] - Rand Corporation ; R-2411-ARPA)
Bibliography: p.
1. EP=2 (Computer system) I. Faught, William S.
II. United States. Defense Advanced Research Projects Agency. III. Series: Rand Corporation. Rand report ; R-2411-ARPA.
AS36.R3 R-2411 [QA76.6] 081s [001.6'4]
ISBN 0-8330-0180-9 79-24404

Published by The Rand Corporation

ARPA ORDER NO.: 3460/3681
0P10 Information Processing
Techniques

R-2411-ARPA
February 1980

EP-2: An Exemplary Programming System

W. S. Faught, D. A. Waterman, P. Klahr, S. J. Rosenschein,
D. M. Gorlin, S. J. Tepper

A Report prepared for

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

PREFACE

Members of Rand's information processing systems research program are currently designing and implementing a software tool called EP (exemplary programming) as a programming aid for non-expert computer users.

This report describes the design and implementation of the current prototype EP system, EP-2. The capabilities, requirements, and constraints relative to the design for a particular task domain and application are also described. The detailed description of the system and the necessary EP-2 facilities should be of interest to designers, implementers, and users of similar systems. The previous version of the system, EP-1, is documented in an earlier Rand report, R-2171-ARPA, *Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition*, by D. A. Waterman, February 1978.

The work reported here is part of an ongoing program of research on software specification issues and artificial intelligence. It has been supported, in part, by the Defense Advanced Research Projects Agency under contract MDA903-78-C-0029.

SUMMARY

One of the main obstacles to effective computer usage is the difficulty of developing software to perform a new task. This report describes the design and implementation of EP-2, an exemplary programming (EP) system, i.e., a system that allows software to be created by example.

The EP paradigm is as follows: The user performs some task on a computer, e.g., transferring a file from one computer to another, retrieving information from a data base, or auditing a data file. The EP system "watches over the user's shoulder", recording the interaction between the user and the system he is using. As the task is being performed, EP constructs an algorithm or model of the interaction. Part of this construction may involve questions to the user or advice from the user. EP then constructs a program (agent) from the model and stores it in a library for subsequent use.

The current version of the EP system, EP-2, is an initial attempt to test this paradigm in the domain of user tasks requiring substantial interactive use of operating systems. It includes the following capabilities: recognizing various prompts that application programs send to the user, constructing sequential actions and simple "while" loops under the user's direction, and using agent call parameters and portions of a program's output in subsequent inputs to the program.

EP-2 has a number of facilities to aid the user. An agent library stores, indexes, describes, and retrieves agents that the user has created. An editing facility makes it possible to correct errors during agent creation. The communication channels (user-system, user-EP, EP-system) can be clearly differentiated by various indentations, or by the use of different colors on a color CRT terminal. A prototype version of EP-2 is currently in operation at The Rand Corporation on a PDP-11 computer linked to a remote PDP-10 computer.

Several issues arise concerning EP's utility in various application areas. For example, some program constructs are extremely difficult for EP-2 to infer. For these, an experimental program-specification facility called the *mind channel* has been developed to provide a "scratchpad" for specifying functions to EP-2. Another problem is that of non-expert users dealing with changing systems that make agents obsolete. This appears to be a special case of the need to control the proliferation and maintenance of software on a system.

Capabilities that would be desirable in future versions of EP are also discussed. The present version is quite limited, having relatively few of the capabilities that can be envisioned in a full-fledged EP system. Nevertheless, these results indicate that the EP paradigm is solid and the current system is usable in a range of applications.

A critique and suggestions for the next version of the EP system are included.

ACKNOWLEDGMENTS

The design of EP-2 has benefited substantially from discussions with Robert Anderson, Randall Davis, Edward Feigenbaum, Frederick Hayes-Roth, Douglas Lenat, and Robert Wesson.

CONTENTS

PREFACE iii

SUMMARY v

ACKNOWLEDGMENTS vii

Section

I. INTRODUCTION 1
- The EP Paradigm 1
- Related Research 1
- EP-2 Research Goals 2

II. THE EP-2 SYSTEM DESIGN 4
- Design Description 5
- The Mind Channel 6

III. THE UTILITY OF EP-2 FOR USER-INTERACTIVE APPLICATIONS 8
- The User's Environment 8
- Specification of Program Constructs 9
- The User's Expertise 9

IV. CONCLUSIONS 11
- Merits 11
- Limitations and Future Directions 11

Appendix

A. DESCRIPTION OF THE EP-2 SYSTEM DESIGN 13

B. AN ILLUSTRATIVE EP-2 SCENARIO 21

C. AN ILLUSTRATIVE EP-2 LADDER SCENARIO 25

D. AN ILLUSTRATIVE INTERACTION WITH THE MIND CHANNEL 30

BIBLIOGRAPHY 37

I. INTRODUCTION

The EP (exemplary programming) system is a set of programs residing on several computers that allows a computer user to construct software by demonstrating examples of the actions that the software is to perform. The EP system generalizes and stores the examples as programs which the user can subsequently execute to accomplish similar tasks.

This report describes the design and implementation of the current version, EP-2, the second in a sequence of EP systems being developed at Rand to explore software specification issues.

THE EP PARADIGM

One of the main obstacles to effective computer usage is the difficulty of developing software to perform a new task. Once a computer user identifies a task, he is faced with the difficult problem of software specification and development.

In the EP paradigm, we have turned the normal programming sequence around. Instead of specifying an algorithm, a program, and an example for debugging, the programmer performs a task, and the EP system infers the algorithm and the program; i.e., the programmer only has to give an example of the desired sequence of actions the program is to perform.

The paradigm is as follows: The user performs some task on a computer, e.g., transferring a file from computer to computer, retrieving information from a data base, or auditing a data file. EP records the interaction between the user and the system he is using. As the user performs the task, EP constructs an algorithm or model of the interaction. Part of this construction may involve a user-EP dialog containing EP queries and user advice. EP then constructs a program from the model and stores it in a library for subsequent use.

RELATED RESEARCH

In many respects, EP relates to earlier research in automatic programming (cf., Balzer et al., 1978; Green, 1977; Heidorn, 1976; Manna & Waldinger, 1978). This research has examined various forms of program specification, including natural language, functional description, and predicate assertions of states or values to be obtained. While these systems form a program from a given set of specifications, EP synthesizes a program from a user-system trace or protocol that represents an example of the task the user wishes EP to synthesize into a program. In this sense, EP is related to the QBE (query-by-example) system developed by IBM for database retrieval. This system retrieves information based on examples of the type of information desired (Girdonsky and Neudecker, 1976).

Synthesizing programs by example has been studied from various viewpoints. Biermann emphasized step-by-step acquisition (Biermann & Krishnawamy, 1976) and algorithms to locate branch points and loops within a program's control flow

(Biermann et al., 1975). Recent work by Bauer (Bauer, 1979) has focused on the need for a specification language within the paradigm for specifying branch conditions and functions—program constructs that are nearly impossible to infer by example. (EP-2 has a similar specification language facility called the *mind channel*; this is discussed in Sec. II.) The EP paradigm itself has been described by Waterman (Waterman, 1978).

The previous EP system, EP-1 (Waterman, 1978), was written in RITA[1] (Anderson & Gillogly, 1976). In its acquisition mode, EP-1 "watched" the user perform some interactive task, queried him about the task, and then produced a set of RITA rules to perform that task. The questioning of the user was necessary because no domain-specific knowledge was built into EP-1.

The basic operation of EP-1 consisted of an initialization phase and an interaction phase. During initialization, the user defined names for each object or type of information (e.g., data types, input/output storage) relevant to the task. During the interaction phase, the user sent messages to the local operating system and the system replied. After each user-system pair of messages, EP-1 would query the user about the value of each object attribute mentioned in the initialization phase. Using these values and the values from the user-system messages, EP-1 would construct a RITA rule of the form "IF condition THEN action". For example:

IF: the name of the current-system is "unix"
and the state of the agent is "use ftp"
and the value of the response contains a {"%"}

THEN: set the name of the current-system to "file transfer program" and set the state of the agent to "give the host name" and set the value of the response to " " and set the reply of the agent to "ftp";

The main drawback to EP-1 was that the user had to supply extensive information after each user-system interaction. In addition, EP-1 had no user aids such as a library of agents or separation of communication channels. As a first test of EP design principles, however, EP-1 was effective.

EP-2 RESEARCH GOALS

Two major questions emerged from the EP-1 research:

1. What types of program-specification applications is the EP paradigm most suited to?
2. What problems are inherent in those applications?

In the EP-2 project we examined three potential application areas: user-interactive tasks, a programmer's scratchpad assistant, and an auditing facility for examining statistical data files. (These applications are further described in Faught, 1979). We then chose one area, user-interactive tasks, for EP-2 implementation. User-interactive tasks are those in which a person uses a computer operating system and its application programs directly—for example, computer network tasks (e.g., file

[1] A set of intelligent terminal agent computer programs developed at Rand.

transfers), operating systems tasks (e.g., file maintenance), data-base retrievals, and edit macros. The major benefit of EP-2 for this application would be to enable non-expert computer users (users who are not programmers) to develop software for user-interactive tasks. A non-expert user could construct software by simply performing his normal task, with EP-2 "watching"; EP-2 would then create a program to accomplish that task.

For this application, EP-2 should be capable of creating agents, storing them in a library, and executing them on request. The user would interact with the local (UNIX) system, with EP-2 watching the interaction. The agents should be sequential programs with variables in the agent call.

The EP-2 system design is described in Sec. II. The utility of the EP paradigm in the application domain is examined in Sec. III; and the report concludes with a critique of the system and suggested improvements for the next version.

II. THE EP-2 SYSTEM DESIGN

This section describes the application area, the basic design of EP-2,[2] and the mind channel, an experiment in building a specification language.

We chose to implement EP-2 for user-interactive tasks in several domains and to simultaneously implement a rudimentary scratchpad for expert users. These tasks seem ideally suited for EP software development: Users need programs to free themselves of repetitive, detailed interactions with application programs; yet writing such programs often cannot be justified because of fast-changing specifications. EP-2 can provide the user, who is most in tune with his own needs and is closest to the problem definition, with a tool to create a library of individualized software.

The domains we chose to examine for EP-2 implementation are operating system commands (in UNIX and TENEX), ARPANET access commands, and two data-base retrieval systems (a publications data base written in RITA and the LADDER ship-deployment data base (Sacerdoti, 1978)). An illustrative LADDER scenario is given in Appendix C.

We can make the paradigm more concrete by describing the flow of information between various components of the current EP-2 system (see Fig. 1). The user interacts with the system as described above, and EP-2 monitors the interaction.[3] User-EP dialog for questions and advice is carried over a separate channel.

When a user wants to create a program, he tells EP-2 to start watching his actions. He then performs a sequence of actions on the computer system. As EP-2 watches, it builds a trace and a model of the interaction. The trace is a verbatim description of the interaction, including user advice. EP-2 keeps it for use with an editing facility for correcting mistakes in the protocol.

From the trace, EP-2 constructs a model of the interaction. The model can be thought of either as a generalization of the trace or as an interpretation (or explanation) of what the user is "really" doing. For example, if the trace contains a sequence of repetitive actions, the corresponding structure in the model might be a loop. User advice both directs the model construction and constrains the space of potential explanations.

Once the protocol is complete, EP-2 finishes constructing the model, perhaps questioning the user to resolve ambiguous interpretations. It can then either generate an agent (a program) in a particular language to perform the task or interpret the model directly to do the task. Trace, model, and agent are also stored in a library for later use.

Because the user must perform his task by interacting with a computer, he must be able to type directions to a device, specifying his actions in real time, using a performance language (such as an executive or monitor command language). Most operating system tasks on interactive computers have such a language; interpreted interactive languages such as INTERLISP also provide this capability. On

[2] See Appendix A for a more complete description of the system design.

[3] In the remainder of this report, the EP-2 system will be referred to as "EP-2"; the phrase "the system" will refer exclusively to the computer system that the user interacts with in order to produce a protocol.

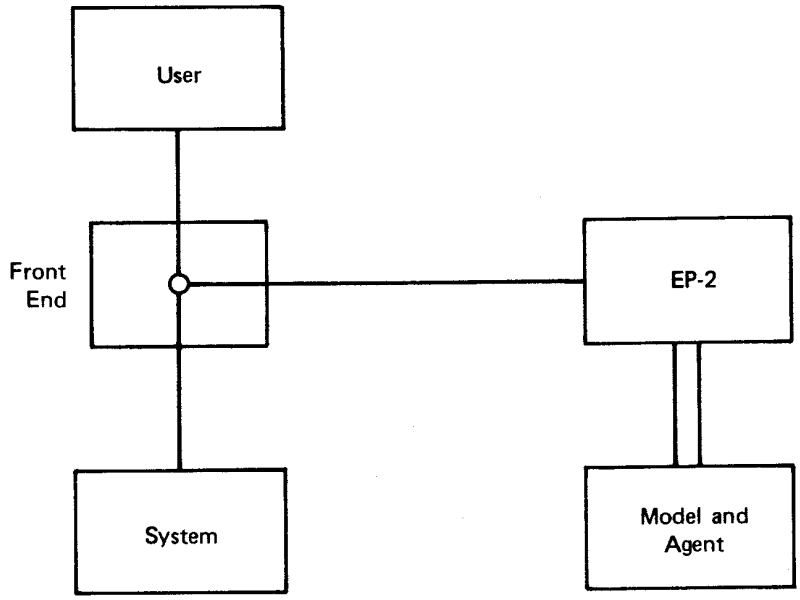

Fig. 1—Components of the EP-2 system

the other hand, compiled languages do not fall within the paradigm. Further, the user may do some actions in his head, such as testing conditions or extracting substrings from system output to use in subsequent system input. EP-2 must acquire these actions, either by inferring them, understanding the user's advice, or having the user demonstrate examples through a mind channel or scratchpad facility (described below). The resulting agent (program) will not necessarily use the system in the same way as the user—e.g., rather than typing (TIMES PI 3) to INTERLISP, it may calculate the value in its own (internal) language. However, for the initial specification of the task, the user must be able to perform all the necessary actions in real time.

A complete transcript of a typical session with EP-2 is reproduced in Appendix B.

DESIGN DESCRIPTION

EP-2 has three basic modes: *dormant*, *create*, and *execute*. When EP-2 is in the dormant mode, the user interacts with the system through the *front end* (which is relatively transparent to him). EP-2 does nothing.

In the create mode, the user interacts with the system through the front end, which marks and sends copies of these messages to EP-2 so that EP-2 can watch. EP-2 records the messages in the *trace*. If the user makes an error and wishes to correct it, he can *edit* the trace. EP-2 then processes the trace (either incrementally or as a whole, depending upon the user's editing) and creates a *model*. A model is a data structure that represents the algorithm the user had in his mind to perform

the actions shown in the trace, the corresponding state of the world after each action, and other actions that the user might take in specific circumstances. Currently, EP-2's create module recognizes various system prompts, constructs straight-line code and simple "while" loops under the user's direction, and uses agent call parameters and portions of system output in subsequent inputs to the system. When EP-2 has finished constructing the model, it stores an interpretable version of the model as a program (or agent) in a *library*, under the name the user provided.

When creating an agent, the user gives EP-2 the name of the agent, i.e., the agent *call*. The call may have *variable* items in it, values for which are subsequently assigned when the user executes the agent. EP-2 stores the agent in the library under the call. The user can then tell EP-2 to *describe* or *delete* the agent named by the call.

When the user calls an agent, EP-2 goes into *execute* mode. It first locates and reads the model from the library; then it interprets the actions in the model, executing them in sequence.

To change modes, the user may communicate directly with EP-2 by pressing a special button on the keyboard. The front end reads this character and prints a specially indented prompt on the user's terminal. The user can then type a command directly to EP-2—for example, to change parameters, to look up or manipulate agents in the library, or to suspend or resume running agents.

THE MIND CHANNEL

An additional feature, the mind channel, was developed as an experiment in program specification. This feature provides the user with another channel for specifying program constructs to EP-2 and a scratchpad for working out details of those specifications. The user-system interaction provides a trace of the user's commands to the system, specified in the performance language; the mind channel provides a language for specifying the user's mental actions. Ultimately, EP-2 should be able to infer from the trace most programs shown by the user (some advice from the user may be required for ambiguous examples). In the meantime, the mind channel allows the user to specify implicit constructs exactly. These constructs include the following:

- *Conditions*—branch conditions that must be true if the agent interpreter is to execute the succeeding actions.
- *Variables*—values in the system response which the agent interpreter is to store in agent variables (and which it may therefore use in succeeding input lines).
- *Return values*—values that the agent should return; i.e., the agent will become a function or procedure that returns a value. This value may be a variable.

The mind channel operates as a scratchpad. The user is initially provided with the agent variables and the most recent system output. He can also define temporary objects, called registers, to store intermediate results. Basically, the user manipulates objects by calling functions with the objects as arguments, creating

new objects (registers). The user can also ask the mind channel to make hypotheses about how to extract subexpressions from larger expressions based on its pattern matcher. When the user has obtained the desired value in a register, he tells the mind channel whether the value is to be used as a branch condition, an agent variable, or a return value. The model constructor then processes this advice by adding a branch condition, a variable, or a return value to the model.

In its current implementation, the mind channel uses INTERLISP to express its functions; thus, the user must be familiar with the INTERLISP programming language. However, there are some operations (e.g., locating a subexpression in a string) that the user can do without knowing INTERLISP.

Appendix D presents an example of the mind channel in operation.

III. THE UTILITY OF EP-2 FOR USER-INTERACTIVE APPLICATIONS

Having implemented EP-2, we examined issues relating to the utility of an EP facility for user-interactive applications. These issues fall roughly into three categories: the user's environment, specification difficulties, and software maintenance by non-expert users. We shall discuss each category in turn.

THE USER'S ENVIRONMENT

User-interactive tasks require EP-2 to act as a willing assistant that is flexible and easy to direct, does not demand substantial guidance, and does not interfere with the user's work. Thus, the first requirement is that the EP-2 environment must be transparent to the user. When the user is not interacting with EP-2, the system response should be as normal as possible; it should not be delayed and it should appear on the screen just as if EP-2 were not there. This is an important consideration in the design of the front end (described below).

EP-2 requires three major information channels: user-system, user-EP, and EP-system. When EP-2 is creating an agent, it monitors all user-system interaction. A switch or filter (the front end) on the user's local system relays input/output messages between the user and the system and sends copies of these messages to EP-2. The user must also be able to communicate directly with EP-2 to give advice and instructions. Finally, EP-2 must be able to send input to the system when executing stored agents. The front end handles all three of these channels in EP-2. Because all of these communication paths are independent of the user's target system, it is most desirable to have the EP system as a front end to the application system.[4]

Several features to assist in user-interactive applications have been incorporated into EP-2. The first is a library of the user's agents (programs), which has several bookkeeping features for maintaining, calling, and describing agents. The second is a "text" feature, by which the user can put text comments into the agent for documentation. These comments, describing the agent's actions, can then be printed to the user as the agent is running. This is particularly useful for tutorial agents and agents that are seldom called. The third feature allows the user to interrupt agent execution, perform actions, and resume EP-2's agent execution. A fourth feature allows the user to interactively correct the protocol, using some form of editing or backup.

EP-2 also incorporates two useful terminal features. In order to keep the various interactions separate and make the output easy to understand, each of the communication channels has a different indentation on the user's terminal. We have also used color terminals with a different color for each communication chan-

[4] The question of whether the EP facility should be built into target systems or should exist as an add-on facility standing between the user and the system has yet to be resolved.

nel and have found this method to be very effective. Separate screens for each channel would be another alternative.

Finally, to make EP-2 easy to communicate with, some kind of shorthand for common user commands, such as special function buttons, should be incorporated in the front end of the system.

SPECIFICATION OF PROGRAM CONSTRUCTS

There are several user-interactive tasks in which it is quite difficult to infer the correct program construct. These constructs include identifying the relevant parts of the system output, constructing functions on the system output, constructing branch conditions, and locating loop boundaries.

In general, these constructs can be identified or determined by one of the following methods:

1. *User specification*—the user is provided with a specification language to describe the desired actions.
2. *Syntax*—the system infers regularities syntactically from a set of examples.
3. *Semantics*—the system contains knowledge about the application domain, e.g., patterns of system output, input command parameters, typical user action sequences.

EP-2 incorporates all three methods. With the mind channel, which emphasizes user specification, the user can specify how to construct functions and branch conditions, although even this activity uses an example-based protocol. A special function in the mind channel uses syntactic methods for locating a substring in a system output string and deriving the function to extract that string. Finally, EP-2 has a set of patterns of system output to parse the output for invariant prompts.

Naturally, it is desirable to reduce EP-2's dependence on the mind channel by increasing the number of syntactic pattern-generation methods and the amount of semantic knowledge in EP-2. However, these capabilities appear to be highly application-dependent and therefore need to be tailored to each set of users' tasks. Additionally, some types of actions seem to be impossible to specify by example and will require that special features be built into the EP-2 system, e.g., waiting for an asynchronous condition to become true.

THE USER'S EXPERTISE

A final set of issues relates to the user's expertise. One of the goals of implementing EP-2 for this particular application was to make interactions with computers easier for non-expert users. The non-expert user will generally either

1. Know how to do a task and construct a program to mimic his actions, or
2. Use agents created by an expert to do his particular tasks.

The first potential problem is that the user may not know the side effects of an agent's actions as he executes it. This does not seem to be an issue in the first case, for the agent will do nothing that the user was not already doing.

In the second case, however, the existence of a tool for creating software easily and quickly can lead to the proliferation of bad software. With an EP-2 system, non-programmers can create new software without following programming standards. This problem is alleviated, to some extent, in the user-interactive task domain by the fact that most interactive systems are verbose about the effect of their actions. Further, the deleterious code must have been executed at least once during the example protocol and thus must have been appropriate in at least one situation. However, code with disadvantageous side effects may still be generated if the user is not careful. It may therefore become appropriate to institute a systemwide certification procedure for public agents.

Another concern is that a system with which an agent has interacted may change, making the agent partially incorrect. The agent will most likely detect the change by failing to obtain the correct response to an input. The set of facilities available to the user for recovery and agent re-creation therefore should be able to print out the actions that the agent would have subsequently taken, together with the original example protocol. The user should then be able to put EP into the create mode and, using the agent's previous actions as the first part of the example protocol, demonstrate the new correct actions.

These recovery facilities will not help the non-expert user, however, if he did not create the agent and does not understand its behavior. In this case, the non-expert has limited options: He can try to patch the agent by comparing the original protocol with the current system output, or he can ask an expert to modify the agent. In a sense, this difficulty is no worse than the problem caused by a program that becomes obsolete as the result of a changing system. However, application systems' user interfaces are likely to change more rapidly than other system features, and EP agents depend substantially upon these interfaces. On the other hand, the resulting changes are typically meant to be human-understandable, so the non-expert may be able to determine the appropriate modification on his own.

To the extent that a user relies upon EP agents for all interaction with systems, he allows the construction of yet another system between himself and the actual machine that is performing his tasks. Barring substantial loss of efficiency, the only dangers we see from this development are those mentioned above: proliferation of nonstandard software and obsolete software plaguing non-expert users who rely on experts' agents. These seem to be problems of software development policy, however, rather than technical or paradigm problems, and they are potentially solvable by dissemination procedures.

IV. CONCLUSIONS

Having described the design and implementation of EP-2, we shall now discuss the relative merits and drawbacks of this system.

MERITS

EP-2 provides the user with a complete set of capabilities for producing simple interactive software by example. It allows him to create agents, edit his examples to correct errors, store the agents in a library that catalogs and indexes them, and execute the agents on request. Each phase of this operation is a distinct step with its own human-engineering aspects. EP-2 enables non-expert computer users to create simple, individually tailored software with only a minimal amount of help from expert users.

The application area we have chosen for implementation of EP-2—user-interactive agents on operating systems and data-base retrieval systems—covers a large set of repetitive, time-consuming tasks that EP-2 could potentially be useful for. Creating agents within this domain is a simple operation. The user tells EP-2 when to start watching and what name to give the agent. He then performs the task as he wants EP-2 to perform it. He tells EP-2 when he is done, and the agent is created and stored with no further action required on the user's part.

The problem of presenting several communication channels to the user on a single computer terminal is solved though the use of differential indentation or color for the different channels. The color feature will be included in future versions of EP.

LIMITATIONS AND FUTURE DIRECTIONS

One obvious drawback to the current EP-2 system is that it is limited to creating agents with sequential actions and simple "while" loops. This severely limits the usefulness of the resulting agent. While we do not wish to minimize the difficulty of synthesizing programs with branches and loops, we feel that EP capabilities can be expanded in future versions, now that the basic system has been designed and implemented. There are a number of algorithms for synthesizing flowcharts which could fairly easily be applied to EP (Biermann, 1976). The main problem is that of limiting the search space of possible flowcharts. A semantic knowledge base (discussed below) would help solve this problem.

Another limitation of EP-2 is that existing agents cannot be modified by example. This seems to be more of an implementation problem than a design problem. It should be no more difficult to modify an existing agent that has been reloaded into EP-2's internal structure than it was to modify the agent as it was being created.

EP-2 has some difficulty with applications that use a CRT display extensively, e.g., two-dimensional screen editors. In order to "understand" an interaction, EP

must (internally) simulate the screen as the user sees it and then interpret the user's actions based on the context of the information on the screen. We have not attempted to implement EP-2 for these applications.

If EP-2 is to become a viable, everyday tool, it must be made to respond more rapidly. One improvement, at least in the execution time of agents, would be to compile the agents on the local system. This could be done only after the agent had been fully developed and may preclude further agent modification.

EP-2 is not portable, and a number of issues must be resolved before EP can be easily accessible to outside users. These issues include front-end processing and knowledge bases geared to the local system. We are currently implementing front ends for other "local" systems.

Finally, during agent creation EP-2 relies heavily on simple syntactic information, e.g., prompts from application programs. The EP-2 knowledge base should be expanded so that EP will have a semantic knowledge of the user-system interaction and can rely on syntactic information as a fallback source. For example, EP needs several knowledge bases of the domain constructs and programming constructs that the user works with.

Of course, semantic knowledge bases are limitless, but there are numerous simple types of information that could easily be put into EP to extend its capabilities. For example, a data base of typical programming constructs, such as "for" loops or sequential file reading, can greatly simplify understanding the branch conditions and data flow in user protocols. With tables of system inputs and outputs, EP could pick out candidate agent variables and track data flows through the agent. Finally, flowcharts of system programs could list the possible system responses at each point and help EP identify potential branches in the agent.

One additional possibility would be to create these data bases, especially the system flowcharts, by example, since the flowcharts will have many of the same features as agents. Once EP synthesized a flowchart, it would remain part of the permanent knowledge base of that domain.

Appendix A

DESCRIPTION OF THE EP-2 SYSTEM DESIGN

This appendix describes the major data structures and components of the EP-2 system design.

TRACE

The trace is a data structure that represents a user-system interaction (or protocol) to accomplish a certain task. It also represents a sequence of actions, or an algorithm, that the user executed for the given task. The trace comprises a list of trace items from the protocol, each of which has a label and a value. The four labels are "user" (input from the user), "system" (output from the system), "agcall" (an agent called by the user while creating an agent), and "advice" (advice from the user to EP concerning the agent being created). The value is a string of alphanumeric characters that are the actual contents of the protocol item. Each "user" or "system" item contains the entire contents of the input or output; i.e., there are no two consecutive "user" items or consecutive "system" items. (There may, however, be consecutive "agcall" or "advice" items.) The front end is responsible for accumulating the items in this manner.

EP currently recognizes several types of advice. "Name" is the call of the agent. "Variables" lists the variable items in the call. "Text" contains English text which the user inserted in the protocol for documentation and which can (optionally) be printed out to the user when EP executes the agent.

The trace represents a complete pass through the algorithm; it therefore will be a linear list of the actions taken with respect to time and will contain no loops or branches.

MODEL

In its simplest form, a model is a data structure that represents the algorithm the user executed to produce the trace. A model is a graph structure of nodes and arcs. Nodes represent states. Arcs have two components: conditions and actions. Several arcs may originate from each node. During execution, EP tests the conditions on all the arcs emanating from a particular node. The arc whose condition is true (there should be no more than one) is the branch to be taken. The arc's action is executed and the state of the agent set to the succeeding node. The current actions are:

- Send <string> to system.
- Send <string> to user.
- Call agent named <string>.
- Read a string from the system up to <prompt>.
- Instantiate a pattern and a list of variables to produce a <string>.

A model can also have global attributes (i.e., a set of attribute names and particular values associated with each name). Current global attributes are the name of the agent, the variables in the agent call, and agent variables (all variables in the agent, including those in the agent call). Agent variables can be manipulated with the mind channel. Arcs and nodes can also have attributes (not currently used). Loops (i.e., "while" loops) are represented by arcs which point to previous states. The model has one distinguished node called the start node and several distinguished nodes marked as end nodes.

CREATE

When the user tells EP to create an agent, EP performs an initialization before watching the user's example. The initialization includes querying the user for the agent call, the variables in the agent call, and an English text description of the agent. EP looks up the name in the library to ensure that it is a legal name (a name that has not already been used and is not an EP command). It then scans the variable list and ensures that all variables are in the call. EP finally creates a new trace, in which it places the agent call, the variable list, and the description as the first three elements.

EP then goes into create mode. It instructs the front end to send EP all of the user-system messages, which it reads and adds to the TRACE. The user may also give EP advice (e.g., "text", "loop", functions from the mind channel), which are also added to the trace.

After EP puts each item into the trace, it constructs the model incrementally.

MODEL CONSTRUCTION

EP has a separate function, called the *model constructor* (MCR), to construct a model incrementally using trace items. MCR has three modes:

1. Incrementally update the existing model to account for the next trace item.
2. Verify that the current model accounts for the next trace item (e.g., during the second time through a loop).
3. Make an entirely new model by starting from the first trace item and iteratively updating the new model.

Normally, MCR uses only the last trace item to update the model. However, whenever the user has edited the trace, MCR constructs an entirely new model using the edited trace. Each of these trace items causes MCR to produce a new arc and an associated action.

Since most of the trace items represent actions that the completed agent is to perform, each of these items causes MCR to produce a new arc and an associated action. We shall describe the MCR processing on each type of trace item.

Name of agent (being constructed). MCR adds the attribute "NAME" to the model and puts the actual agent call in the attribute's value.

Variables in agent call. MCR puts these variables on the value of the attribute "AGVARS" of the model. It also puts them on the list of agent variables. (Other variables could originate from system output strings.)

User input to system. MCR searches a data base of user input patterns (string patterns with literals and variables) for the input string patterns. The explanation of the user's action is that he evaluated one of these patterns with certain values for the variables; this evaluation returned a string. He then typed the string to the system as an input.

If MCR finds no pattern, it must construct one. To do so, MCR gives EP's pattern generator two parameters: the input string and a list of the agent variables. The pattern generator searches for any instances of agent variables. It then constructs a string pattern and replaces each agent variable with a local pattern variable.

MCR now has a pattern that matches the input, whether retrieved or newly constructed. MCR then replaces all of the local pattern variable names with the agent variable names that they matched. During agent execution, the agent interpreter, if called with this pattern and the original agent variables, will compute the original string. We will refer to this entire process of locating (or generating) a pattern and replacing its variables as "instantiating" a pattern.

MCR then forms the action (SENDSYS <pattern>) and places it as the action of a newly-created arc and node.

System output to user. MCR matches the output string (which may be several lines) against a list of known prompts to find which prompt the output ended with. If EP finds no known prompt, it extracts the last two characters from the output and uses that substring for a prompt. MCR creates a string pattern {"any number of characters followed by" <prompt>}. MCR then forms the action (READSYS <pattern>) and places it into a new arc.

Agent call (of a subagent being executed.) MCR uses the input string to instantiate a pattern, similar to user input to the system. It then forms the action (AGCALL <pattern>) and places it in a new arc.

Text. Again, MCR uses the input string to instantiate a pattern. It then forms the action (AGUSER <pattern>) and places it in a new arc.

Loop. The user tells EP he is looping back to a previous action. MCR examines the trace for a previous action matching the input string and constructs an arc in the model from the previous action to the matched action. The user must later use the mind channel to create a branch condition out of the loop.

When the user indicates that the protocol is complete, MCR tags the last node as an end node (an exit point from the agent). It then stores both the model and the trace in the library.

THE MIND CHANNEL

The mind channel provides a scratchpad for debugging individual lines of code. To enter the mind channel, the user types a special command to EP ("MC"). The mind channel has a list of registers (named TMP1, TMP2, . . . , TMPn) for manipulating expressions. Each time the user types an expression, the mind channel creates a new register containing the value of the expression. There is also a special name (currently, a semicolon) denoting the most recently created register. (The mind

channel discards the register list whenever it is exited.) The user can print the list of registers, their values, and the composite functions of how they were derived.

When the user enters the mind channel, it stores the previous system output line in TMP1 (the first register). The user can also load all of the agent variables (in the call and from other mind-channel interactions) with the variables command.

The user can manipulate expressions stored in the registers by typing INTERLISP expressions with either a semicolon or TMPn as arguments. For example, the following are legal mind-channel expressions:

```
(CAR ;)
(CAR TMP2)
(PLUS ; 2)
(MKSTRING ;)
(MKATOM TMP3)
(CONS ; TMP4)
```

If the function has only one argument, the user can omit the argument and the parentheses. The mind channel will assume that ";" is the argument.

When the user has isolated and/or constructed the desired expression, he tells the mind channel whether EP is to consider it as a new variable, a condition, or a return value.

The MCR then processes this advice as follows:

Condition. MCR adds a new arc to the model with a null action and the composite function as the arc's condition.

Variable. MCR makes a new agent variable and adds an arc whose action is to set the variable to the value of the evaluated composite function.

Return value. MCR adds a new arc to the model whose action is to set the special variable RETURNVAL to the value of the evaluated composite function.

The mind channel has the ability to make hypotheses about how to extract subexpressions from larger expressions based on its pattern matcher. The LOC command invokes EP's pattern generator on the main expression, using the subexpression as advice on where to locate the variable. For example, if the current expression is "18 Feb 1979", the user can type "(LOC 18)" and the pattern generator will create a pattern for the current expression which, when applied by the pattern matcher to the expression, will return the correct value. The mind channel also builds the function to apply the pattern matcher to the pattern and the expression.

Of course, the register list is simply a scratchpad. Thus, if the user makes a mistake, or if the mind channel makes a wrong hypothesis, the current value, ";", can be reset from a previous value and the incorrect result (in a register) can be ignored.

EDIT

EP has an editing facility to allow the user to correct errors in the protocol. EP considers the trace as the definitive copy of the protocol. Therefore, if the user wishes to edit the example, he must edit the trace. Since EP stores the trace separately as a list of trace items, edit processing is rather straightforward.

Currently, there are two editing functions: backup and display-trace. Backup

deletes the last trace item, and display-trace displays the entire trace on the user's terminal. Thus, if the trace has five lines and the user determines that the fourth line is in error, he simply types "backup" twice, displays the trace to verify that he has deleted enough entries, and then resumes adding to the trace by example.

Edit commands do not alter the model directly; i.e., EP does not try to deduce the effect on the model of each edit command. Instead, whenever the user alters the trace with an edit command, MCR deletes the current model. When the user resumes adding to the trace, MCR constructs a new model from the edited (and presumably correct) trace by starting from the first trace item and iteratively updating the new model.

EXECUTE

When the user calls an agent, EP looks up the agent name in the library. If it finds the name, it reads in the agent, sets the appropriate agent variables to the values in the call, and starts the agent interpreter.

The function of the agent interpreter is to execute the model by following arcs from nodes, testing conditions on arcs, and performing actions. The agent interpreter works on the model. It has a pointer (CNODE) to a node in the model which it uses to keep track of where it is processing in the model. It first sets the current pointer CNODE to the start node. It then finds all arcs emanating from that node and tests all arc conditions to find one that is true. It then performs the action on that arc and sets CNODE to the node that the arc points to. This continues until CNODE contains an end node.

The interpreter's progress can be interrupted in several ways. If there are no conditions on the arcs emanating from a node that are true, the interpreter halts and sends a message to the user. The user can also interrupt the interpreter manually with the "interrupt" command. Finally, the agent may "timeout", waiting for a particular system output (to be described later). When any of these interrupts occurs, the interpreter saves the CNODE pointer so that execution may resume from that point. The interpreter's processing on each action is described below.

Agent input to system. (SENDSYS <pattern>): The agent interpreter instantiates the agent variables in the pattern and evaluates the contents of the pattern to produce a string. It then sends the string to the system.

Agent message to user: (SENDUSER <pattern>). The interpreter evaluates the pattern (as for SENDSYS) and produces a string. EP sends the string to the user with a special indentation on the user's terminal to indicate that the message is from the agent. The "text" feature is implemented with this action.

Agent call (of a subagent): (AGCALL <pattern>). The interpreter evaluates the pattern (as for SENDSYS) and produces a string. The agent interpreter calls itself recursively on the agent named by the string. This action is not complete until the agent interpreter has finished processing the subagent.

System output to agent: (READSYS <pattern>). This action attempts to read output from the system until one of two events occurs. If the pattern matches the accumulated output, the action is complete. If a certain amount of time (which the user can set) passes and the pattern does not match, EP considers the action to have failed. (This is called "timeout".) In this case, either of two things can happen. If

a special flag (INTONTIMEOUT, which the user can set) is set to true, the agent will interrupt. If the flag is set to false, the action will return and the interpreter will continue as though the pattern had been matched.

There are several reasons for these options. If the agent knows what the correct output should be and the system prints this output quickly, the agent should continue processing immediately. If the agent does not know what the output should be (because the MCR did not know the prompt), then the system may have typed the correct output, in which case the user may want EP to continue with agent execution. If, however, the user wants to be cautious, he can specify that the interpreter is to interrupt whenever it does not find the anticipated output. Finally, the system may be very slow, in which case the correct output may be on its way, but may not yet have arrived. Thus, the user should be able to set the waiting time for system output.

Several features (not implemented in EP-2) would make solutions to these problems more viable. First, if EP had specific patterns for all the possible outputs to any input, it would know when it read the correct output and could determine when the system was typing the correct output but had not yet typed all of it. Second, if EP knew the average response time of the system, it could dynamically set the waiting time. Finally, if EP knew which system responses were chronically slow, it could wait an extra measure of time for them. Most of these improvements will have to await an EP data base of knowledge about systems in the task domain.

LIBRARY

The function of the library routines is to store, catalog, and retrieve agents and their traces. The agents and traces are stored on a TENEX directory, each with a unique name. Each user also has a library index file which lists all of the agents in his library. EP reads the library index file when it is first run and stores the library index internally. All of the library routines access this library index.

The main library routines will store, retrieve, describe, and delete agents in a straightforward manner. All of these routines use one LOOKUP routine, which takes a string as input and searches through the library index for all agents that (at least) partially match that string. LOOKUP returns this list to the calling library routine. For example, if the user wishes to delete an agent and the LOOKUP routine returns several candidate agents, he must choose among them. If the user wishes to describe an agent (print out the agent call, variables in the call, and the English text description supplied by the agent's creator), all candidate agents will be described.

FRONT END

The front end serves as (1) a monitor or filter through which EP watches the user-system interaction, (2) a program that the user can talk to and request actions from, and (3) an interface between the local UNIX system and TENEX INTERLISP.

As a monitor, the front end sends to EP everything that the user types to the operating system and everything that the system outputs to the user. It sends data typed by the user to the system and output from the system back to the user, so

that the user appears to be talking directly to the system. He is unaware of the front end except for degradation in echo time.

Whether or not the front end sends this information to EP is under EP's control through a set of EP-to-front-end commands. Basically, the front end sends this input whenever EP is in the create or the execute mode. In the dormant mode, EP sees none of this interaction.

Additionally, the user may wish to talk to EP directly. The front end handles this interaction for EP over the same front-end-to-EP channel. If the user types a particular (reserved) key, the front end will print a special indented EP prompt ("[EP:]"), read one line of user input, and send the input to EP flagged as a user-EP input. EP can then type its replies on the user's terminal through the front end. EP can also initiate this dialog with a special EP-to-front-end command that locks the user's keyboard, types EP's query to the user, and unlocks the keyboard for the user's reply. The front end sends this reply to EP as a user-EP input.

Finally, EP may "type" its own input to the system. Since EP runs on a different machine, the front end manages this insertion of data. EP simply flags a message from EP to the front end as an EP-system message, and the front end sends the message to the system.

One other communication path is from a running agent to the user for the "text" feature. In order to distinguish this "text" from system output and from normal EP output, the front end indents the output on the user's terminal and encloses it in square brackets.

As an interface between two operating systems, the front end must reside on the system that the user considers to be "local". Thus, a new front end must be developed for each new "local" system. The same EP module, however, can be used for all front ends.

SPECIAL FACILITIES FOR SYSTEM MAINTENANCE

We have designed several facilities that assist in EP-2 maintenance.

The first of these is an automatic saving and cataloging of EP errors (bugs in the EP software). Because EP-2 was designed in modules, it was relatively simple to place error traps around the basic modules. When EP encounters one of these errors, it automatically writes an error message with diagnostics onto a special error disk area. The maintenance staff periodically scans these error messages and takes measures to correct the error. EP also writes a message to the user to warn him of possible random results.

The second facility was developed to forestall the potential lack of upward compatibility of agent and model design. After an initial version of EP-2 was released, users proceeded to create agents to perform some of their repetitive tasks. When we released the next version of EP-2, we had redesigned the format of agents and made the previously constructed agents incompatible.

To counteract future problems stemming from redesigned agents, we decided to program EP-2 to save traces as well as agents, since the trace structure was simple enough to anticipate that its design would change infrequently. We then implemented an additional EP command called "tracetomodel", which, when called by the user with a trace name, reads in the trace and proceeds to create a model

as if the user had just created the trace by example. The new model then replaces the old one (with the new model structure) and is compatible with the new design.

Of course, there was also an initial period in which there were a number of agents with no traces. We therefore also implemented a "modeltotrace" feature which runs a model and creates and stores a trace. These two features have allowed us a freer hand in upgrading the design of EP-2.

IMPLEMENTATION

To implement the above design, we coded the main body of EP in INTERLISP running on a PDP-10 computer using the TENEX operating system. EP resides in a single user process of 300 pages[5] (plus the INTERLISP overhead of 256 pages).

We implemented the front end in the C language on a PDP-11 computer using the UNIX operating system. The front end resides in eight user processes, most of which are two blocks long, with the exception of the main process, which is 44 blocks (a block is 512 eight-bit words). The user-system interaction rate is rapid; the front end causes little slowdown. Thus, there is little overhead when EP is dormant. (We are currently implementing another front end in SAIL for use on a TENEX system.)

EP and the front end communicate through a dual ARPANET data connection from program to program (i.e., not using a TELNET server at either end). Communication on this channel is in multiple-character messages (generally from 5 to 50 characters). Thus, the data connection is as rapid as can be expected.

As is typical of an experimental system, EP's processing slows down the user's interaction with the system, in this case by (approximately) a factor of two. The slowness seems to originate from two sources. The first is the slowness of INTERLISP when the TENEX machine is heavily loaded, and the second is the long communication link between the operating system and EP. Several processes must handle a system output before EP sees it: the front-end pty process, the front-end main process, the front-end net process, the local UNIX ARPANET server, the ARPANET itself, the remote TENEX ARPANET server, INTERLISP, and EP's system output buffer. Once EP decides that the system output is complete and the next input can be sent, the input must traverse a similar path back to the system. This slowdown could be eliminated by implementing EP on the local computer, closer to the user-system interaction. Because we are still experimenting with EP's design and features, however, this improvement has low priority.

[5] One page is 512 36-bit words.

Appendix B

AN ILLUSTRATIVE EP-2 SCENARIO

The following is a scenario of the current EP system in operation. The user, interacting with the UNIX operating system, wishes to create a program to print a file that is on a remote system on the local printer. The program must retrieve the file from the remote system using the file transfer protocol (ftp), print it, and delete it.

In this protocol, the unix prompt is "%". The user-system interaction is at the left margin; the EP-user interaction is indented 20 spaces; and the annotation is in braces.

```
% ep
Telnetting... logging in... starting EP...

                                {The user starts the EP system.}

%
                        The GOTO button is control-P
                        For help, type GOTO and "help<carriage return>".

%
                                {EP displays a standard "help"
                                 message.  To get EP's attention,
                                 the user types a special character,
                                 <control-P>.}

                        [EP]: create

                                {The user tells EP to create an agent.}

                        Name of agent: print do.doc from ecl

                                {EP responds by asking for the name of the
                                 agent.}

                        Variables in agent call: do.doc

                                {EP then asks the user what variables are
                                 in the agent call.}

                        Making new var: VAR1, with value: do.doc

                                {EP makes variables for later instantiation.}

                        Describe agent...
                        Text: This agent retrieves the file do.doc
```

```
                    Text: from ecl and prints it locally.
                    Text:

                                {The user can give a description of the agent
                                 as a form of documentation.}

                    EP is watching
                     -EP waiting-

                                {EP has finished acquiring the initial
                                 information about the agent, and now watches
                                 the user as he interacts with the system.}

% ftp ecl

                                {The user starts up FTP, and logs in to
                                 the remote server.}

Connections established.

300 USC-ECL FTP Server 1.44.11.0 - at WED 12-JUL-78 14:12-PDT
> user faught
330 User name accepted. Password, please.
> pass
Password:
230 Login completed.
> retr do.doc
                                {The user tells FTP what remote file he
                                 wishes to retrieve.  EP recognizes the
                                 file name as being a variable in the agent
                                 call.  EP creates an action in the model to
                                 evaluate the variable at run time.}

localfile:  temp.bak

                                {The user selects a temporary file name as
                                 the local destination.}

255 SOCK 3276867075
250 ASCII retrieve of <FAUGHT>DO.DOC;2 started.
252 Transfer completed.
> bye
231 BYE command received.
% print temp.bak

                                {The user prints and deletes the local file.}

% del temp.bak
temp.bak

%
```

```
                    [EP]: end
                                {The user tells EP that he is done with the
                                 task.}

                    End agent called: print do.doc from ecl
                    End agent construction
                    Agent stored in library
                    Trace stored in library
                     -EP dormant-
%

                                {The agent is now available for use.  The
                                 user calls it on a different file.}

                    [EP]: print pattern from ecl

                    -Calling: print pattern from ecl
%
          [This agent retrieves the file pattern
          from ecl and prints it locally.]

                                {The agent starts running by first printing
                                 the description of its task.}

% ftp ecl

                                {The agent types to the system, just as
                                 the user did.}

Connections established.

300 USC-ECL FTP Server 1.44.11.0 - at WED 12-JUL-78 14:15-PDT
> user faught
330 User name accepted. Password, please.
> pass
Password:
230 Login completed.
> retr pattern

                                {The agent tells FTP which remote file is
                                 to be retrieved.  The file "pattern" is
                                 instantiated from the agent call.}

localfile:  temp.bak
255 SOCK 3276867075
250 ASCII retrieve of <FAUGHT>PATTERN.;1 started.
252 Transfer completed.
> bye
231 BYE command received.
% print temp.bak

                                {The agent prints and deletes the local file.}
```

```
% del temp.bak
temp.bak

%
                    -Ending agent: print pattern from ecl
                     -EP dormant-
%
```

Appendix C

AN ILLUSTRATIVE EP-2 LADDER SCENARIO

The following is an example of a data-base retrieval task using the LADDER data-base system. The agent uses the LADDER system to print the status of all ships within 200 miles of the default location (currently NORFOLK). The agent calls five subagents in sequence which perform the following operations:

1. Logs into a remote system (SRI-KL) and starts up the LADDER system.
2. Defines the term "opstatus" in LADDER, referring to the current position, fuel status, state of readiness, and commanding officer of a ship.
3. Types the specific retrieval request for ships within 200 miles.
4. Exits from the LADDER system.
5. Formats, prints, and saves the transcript.

```
% ep
Telnetting... logging in... starting EP...
%
                    The GOTO button is control-P
                    For help, type GOTO and "help<carriage return>".

%
                    [EP]: ships status 200 miles

                    -Calling: ships status 200 miles
%
          [This agent uses the LADDER system to
          print the status of all ships within 200 miles
          of the default location (currently NORFOLK).
          Types of ships recognized by LADDER are:
           ships submarines carriers cruisers]
%
          [PHASE 1: Start the LADDER system.]
%
                    -Calling: ladder
%
          [This starts the LADDER system at SRI-KL.]
%
          [Telnet to SRI with a TEE for saving results of
          this session.]
% tn sri|tee ladder.temp
Open

 SRI-KL, TOPS-20 Monitor 101B(116)
 System shutdown scheduled for Mon 18-Sep-78 00:01:00,
 Up again at Tue 19-Sep-78 04:00:00
 There are 43+8 jobs and the load av. is  3.94
```

```
@
          [Login as a LADDER user.]
@login
(user) fhollister
(Password)
(account)
 Job 84 on TTY251 14-Sep-78 17:09
 Previous login: 13-Sep-78 15:46 from host RAND-UNIX
 [There are 3 other jobs in group DA]
@
          [Logged in; now start LADDER.]
@ladder

        Language Access to Distributed Data with Error Recovery
                        -- SRI International --

Please type in your name:
          [User's name (for repeated requests).]
Please type in your name: epdemo
When you are finished, please type DONE.
This will close the Datacomputer files.
Do you want instructions? (type FIRST LETTER of response)
          [Bypass the intructions.]
Do you want instructions? (type FIRST LETTER of response) No
Do you want to use 2 Data Computers?
          [Use only 1 Data Computer.]
Do you want to use 2 Data Computers? No
Do you want to specify a current location (default = Norfolk)?
          [Use the default location (for this user).]
Do you want to specify a current location (default = Norfolk)? No
Do you wish distance/direction calculations to default to GREAT CIRCLE,
or RHUMB LINE? (you can override by specifying in the query)
          [Use the Great Circle calculation.]
or RHUMB LINE? (you can override by specifying in the query) Great Circle

1_
          [Suppress the Data Computer specification.]
1_set verbosity to be -1
                          PARSED!
-1
2_
          [LADDER is now waiting for English input.]
2_
                    -Ending agent: ladder

2_
          [PHASE 2: Define special terms.]
2_
                    -Calling: ladstatus
2_
          [This defines opstatus in LADDER.]
```

```
2_define (what is the opstatus of jfk)
...like (what is the current position, fuel status, state of readiness,
and commanding officer of jfk)
...

PARSED!

  WHAT IS THE CURRENT POSITION FUEL STATUS STATE OF READINESS AND COMMANDING
OFFICER OF JFK



            PARSED!

For SHIP equal to KENNEDY JF, give the POSITION and DATE and PCFUEL and READY
 and RANK and NAME.


May LIFER assume that "CURRENT POSITION FUEL STATUS STATE OF READINESS AND
COMMANDING OFFICER" may always be used in place of "OPSTATUS"? Yes

<RELN>  =>  OPSTATUS
(OPSTATUS)
3_
          ["Opstatus" is now defined.]
3_
                    -Ending agent: ladstatus

3_
          [PHASE 3: Type the specific retrieval request.]
3_
          [LADDER processes the request by interacting
          with the Data Computer.]
3_what is the opstatus of all ships within 200 miles
                                                    FROM NORFOLK
                                                    PARSED!


For great circle distance to 37-00N, 76-00W less than or equal to 200, give the
POSITION and DATE and PCFUEL and READY and RANK and NAME and SHIP.

 Connecting to Datacomputer at CCA:
>> ;0031 780915001150   IONETI: CONNECTED TO SRI-KL-30700010
>> ;J150 780915001151   FCRUN: V='DC-5/01.00.13' J=7 DT='THURSDAY, SEPTEMBER 14,
 1978 20:11:51-EDT' S='CCA'
>> ;J200 780915001151   RHRUN: READY FOR REQUEST

*> Set parameters
*<      Exit
.CCA: ^Z
*> Set parameters
*< V    Verbosity (-1 to 5):   -1......
```

```
POSITION DATE           PCFUEL READY RANK        NAME         SHIP
37-00N, 076-00W 17Jan76, 1200 0      C5    CAPT HALSEY W    AMERICA
37-00N, 076-00W 17Jan76, 1200 100    C1    CAPT BROWN A     SARATOGA
37-00N, 076-00W 17Jan76, 1200 100    C1    CDR  SMITH R     STURGEON
37-00N, 076-00W 17Jan76, 1200 100    C1    CDR  COHEN X     WHALE
37-00N, 076-00W 17Jan76, 1200 100    C1    CDR  HIGH J      TAUTOG
37-00N, 076-00W 17Jan76, 1200 100    C1    CDR  DAUGHERTY R GRAYLING

4_
                    -Calling: beep
4_
          [This beeps the user's terminal.]
4_
                    -Ending agent: beep

4_
                    -Ending agent: ships status 200 miles
                     -EP dormant-
4_
                    [EP]: ladexit

                    -Calling: ladexit
4_
          [Exiting LADDER, back to unix...]
4_
          [PHASE 4: Exit from the LADDER system.]
4_done
        PARSED!
.Thank you
@k
[Confirm]
 System shutdown scheduled for Mon 18-Sep-78 00:01:00,
 Up again at Tue 19-Sep-78 04:00:00
Logout Job 84, User FHOLLISTER, Account DA, TTY 251, at 14-Sep-78 17:14:02
  Used 0:0:12 in 0:4:56
%
          [PHASE 5: Format, print, and save the transcript.]
%
                    -Calling: ladsave
%
          [This saves the results of each LADDER run
          by appending them on the file LADDER.RESULTS
          and printing them on the RCC computer.]
%
          [Delete cr's and DEL's.]
% tr -d "\015\177" < ladder.temp > ladder.temp1
%
          [Use the ED editor to delete unwanted parts
          of the LADDER protocol.]
% ed ladder.temp1
3211
/^\-1/p
```

```
-1
1,.d
/._done/p
?
.,$d
w
?
q
%
          [Append the results to the archive file.]
% cat ladder.temp1>>ladder.results
%
          [Print a hard copy of today's results.]
% print ladder.temp1
%
          [Remove the temporary files.]
% del ladder.temp ladder.temp1
ladder.temp
ladder.temp1

%
                    -Ending agent: ladsave

%
          [You are now talking to unix.]
%
                    -Ending agent: ladexit
                     -EP dormant-
%
```

Appendix D

AN ILLUSTRATIVE INTERACTION WITH THE MIND CHANNEL

The following is an example of agent creation using the mind channel.

The user has a file of accounts. Each text line of the file contains the account number, account name, and current balance. In this example, the first line is

123 JONES, WALTER K. 1900

The user wishes to create an agent that will select all accounts with balances greater than or equal to 1000 and write them out to a separate file. The user has access to an interactive interpreted language with which he can read, examine, and write files line-by-line. Typical commands in the language are:

INFILE—open a file for reading.
OUTFILE—open a file for writing.
READALINE—read one line from the input file.
PRINT—print a line to the output file.

The mind channel provides the facility for manipulating the contents of lines for conditional actions.

In this protocol, the prompt is "*". The user-system interaction is shown at the left margin; the EP-user interaction is indented 20 spaces; and the annotation is in braces.

```
@
                    [EP]: create

                    Name of agent:
                    -> LARGEBAL TESTFILE
                                {The user has a TESTFILE of examples
                                 for agent creation.}
                    Variables in agent name:
                    -> TESTFILE

                    Describe your agent...
                    Text: This agent selects balances of greater than 1000

                    Text: from file TESTFILE and writes them to TESTFILEOUT.

                    Text:

                    -EP is watching-
                     -EP waiting-
@BAS

                               {The user starts up the applications system.}
```

```
INTERLISP-10   7-DEC-78 ...

Good afternoon.

(<FAUGHT>BAS.SAV;1 . <LISP>LISP.SAV;127)
* (INFILE 'TESTFILE]
T
* (OUTFILE 'TESTFILEOUT]
T
* (SETQ A]
NIL
* (SETQ A (READALINE]
(A reset)
"123 JONES, WALTER K.    1900"
                              {The user reads the first input line...}
* (PRINT A]
"123 JONES, WALTER K.    1900"
                              {... and writes it in the output file.}
*
                    [EP]: loop READALINE

                    Run the loop? n

                              {The user tells EP to loop back to the
                               action containing the string READALINE.
                               EP asks whether to continue running the
                               agent created automatically.  The user
                               says "no".}
                     -EP waiting-
* (SETQ A (READALINE]
(A reset)
"124 SMITH, ROBERT        688"
*
                    [EP]: MC
                              {The user enters the mind channel.}

                    Adding a branch - line not verified: System:
                    (A reset)
                    "124 SMITH, ROBERT        688"
                    *
                              {EP detects a new branch being taken
                               by noting that it cannot verify that
                               an equivalent system response was received
                               to the user's action.}
                      MC forms:
                    ; TMPn
                    Quit  eXit
                    Print  Remember  Variables  Condition
                    LOC val   or   (LOC val)
                    (FN ; TMPn lit)
                    CAR
                    Any forms to be filled in: precede with "T "
```

```
            {The mind channel prints a summary of
             its commands.}

Mc: LOC 688
            {The user asks the mind channel to locate
             the string "688" in the system output.}

Finding "688" in "
(A reset)
"124 SMITH, ROBERT        688"
* "
TMP2 = "688"   Fn: (CADR (ASSOC (QUOTE V1) (XMATCH
RESPONSE (QUOTE ((: BLANK) (& ANY V2) (* () (& ALPHANUM
V3) (& BLANK) (& ALPHANUM V4) (* )) (& ANY V5) (* ") (&
ALPHANUM V6) (& BLANK) (& ALPHANUM V7) (* ,) (& BLANK) (&
ALPHANUM V8) (& BLANK) (& ALPHANUM V1) (* ") (& ANY V9)
(* * ) (: BLANK))))))

            {The mind channel locates the string "688",
             sets the value of the register named "TMP2"
             to it, and remembers the function used to
             derive the string "688" from the original
             string. }

   ; = "688" : TMP2

            {The register named ";" points to TMP2 also.
             The user refers to it subsequently.}
Mc: MAKNUMM ;

            {The user converts the string "688" into a
             number with the MAKNUMM function.  Note
             how the MAKNUMM function embeds the
             previous function specification in the
             function below.}

TMP3 = 688   Fn: (MAKNUMM (CADR (ASSOC (QUOTE V1) (XMATCH
RESPONSE (QUOTE ((: BLANK) (& ANY V2) (* () (& ALPHANUM
V3) (& BLANK) (& ALPHANUM V4) (* )) (& ANY V5) (* ") (&
ALPHANUM V6) (& BLANK) (& ALPHANUM V7) (* ,) (& BLANK) (&
ALPHANUM V8) (& BLANK) (& ALPHANUM V1) (* ") (& ANY V9)
(* * ) (: BLANK)))))))

   ; = 688 : TMP3

Mc: LESSP ; 1000

            {The user tests to see that the value is
             less than 1000.}

TMP4 = T   Fn: (LESSP (MAKNUMM (CADR (ASSOC (QUOTE V1)
(XMATCH RESPONSE (QUOTE ((: BLANK) (& ANY V2) (* () (&
ALPHANUM V3) (& BLANK) (& ALPHANUM V4) (* )) (& ANY V5)
```

```
(* ") (& ALPHANUM V6) (& BLANK) (& ALPHANUM V7) (* ,) (&
BLANK) (& ALPHANUM V8) (& BLANK) (& ALPHANUM V1) (* ") (&
ANY V9) (* * ) (: BLANK))))))) 1000)

   ; = T : TMP4
Mc: C
                {The user tells the mind channel to construct
                 a condition from the function.  The mind
                 channel inserts the function and a label as
                 advice into the trace.  EP subsequently
                 attaches the corresponding condition to the
                 current arc in the model.}

New condition: (LESSP (MAKNUMM (CADR (ASSOC (QUOTE V1)
(XMATCH RESPONSE (QUOTE ((: BLANK) (& ANY V2) (* () (&
ALPHANUM V3) (& BLANK) (& ALPHANUM V4) (* )) (& ANY V5)
(* ") (& ALPHANUM V6) (& BLANK) (& ALPHANUM V7) (* ,) (&
BLANK) (& ALPHANUM V8) (& BLANK) (& ALPHANUM V1) (* ") (&
ANY V9) (* * ) (: BLANK))))))) 1000)
Mc: X

Leaving MC
                {The user exits the mind channel, having
                 constructed a condition for this branch
                 in the loop.}
 -EP waiting-
*
[EP]: loop READALINE

Run the loop? n

                {The user tells EP to again loop back to the
                 action containing the string READALINE.}

 -EP waiting-
* (SETQ A (READALINE]
(A reset)
"STOP"
*_
[EP]: MC
                {The user enters the mind channel for
                 a condition to end the loop.}

Adding a branch - line not verified: System:
(A reset)
"STOP"
*
  MC forms:
; TMPn
Quit  eXit
Print  Remember  Variables  Condition
LOC val   or   (LOC val)
(FN ; TMPn lit)
```

```
CAR
Any forms to be filled in: precede with "T "
Mc: LOC STOP

            {The user tells the mind channel to single
             out the substring "STOP".}

Finding "STOP" in "
(A reset)
"STOP"
 "
TMP2 = "STOP"   Fn: (CADR (ASSOC (QUOTE V1) (XMATCH
RESPONSE (QUOTE ((: BLANK) (& ANY V2) (* () (& ALPHANUM
V3) (& BLANK) (& ALPHANUM V4) (* )) (& ANY V5) (* ") (&
ALPHANUM V1) (* ") (& ANY V6) (* _) (: BLANK))))))

   ; = "STOP" : TMP2
Mc: EQUAL ; "STOP"

            {The first part of the input line must equal
             the substring "STOP".}

TMP3 = T   Fn: (EQUAL (CADR (ASSOC (QUOTE V1) (XMATCH
RESPONSE (QUOTE ((: BLANK) (& ANY V2) (* () (& ALPHANUM
V3) (& BLANK) (& ALPHANUM V4) (* )) (& ANY V5) (* ") (&
ALPHANUM V1) (* ") (& ANY V6) (* _) (: BLANK))))))) STOP)

   ; = T : TMP3
Mc: C
            {The user makes in a condition.}

New condition: (EQUAL (CADR (ASSOC (QUOTE V1) (XMATCH
RESPONSE (QUOTE ((: BLANK) (& ANY V2) (* () (& ALPHANUM
V3) (& BLANK) (& ALPHANUM V4) (* )) (& ANY V5) (* ") (&
ALPHANUM V1) (* ") (& ANY V6) (* _) (: BLANK))))))) STOP)
Mc: X

Leaving MC
 -EP waiting-
*

[EP]: text

Text: Now close the files.

Text:

 -EP waiting-
* (CLOSEF 'TESTFILEOUT]
<FAUGHT>TESTFILEOUT.;1
* (LOGOUT)
@
[EP]: end
```

```
                    -End agent called: LARGEBAL TESTFILE

                    ----------------------
                    -Agent stored in library
                    -Trace stored in library
                     -EP dormant-
@
                                {Now the user runs the agent on a data file.}

                    [EP]: LARGEBAL ACCTS

                    -Calling: LARGEBAL ACCTS
@
          [This agent selects balances of greater than 1000
           from file ACCTS and writes them to ACCTSOUT.]
@BAS


INTERLISP-10   7-DEC-78 ...

Hello.

.bp
(<FAUGHT>BAS.SAV;1 . <LISP>LISP.SAV;127)
* (INFILE 'ACCTS]
T
* (OUTFILE 'ACCTSOUT]
T
* (SETQ A]
NIL
* (SETQ A (READALINE]
(A reset)
"123 JONES, WALTER       1900"
* (PRINT A]
"123 JONES, WALTER       1900"
* (SETQ A (READALINE]
(A reset)
"124 SMITH, ROBERT        688"
* (SETQ A (READALINE]
(A reset)
"126 SMITH, JACK            8"
* (SETQ A (READALINE]
(A reset)
"130 WILLIAMS, D.        3427"
* (PRINT A]
"130 WILLIAMS, D.        3427"
* (SETQ A (READALINE]
(A reset)
"STOP"
*
          [Now close the files.]
* (CLOSEF 'ACCTSOUT]
<FAUGHT>ACCTSOUT.;8
```

```
* (LOGOUT)
@
                    -Ending agent: LARGEBAL ACCTS
                     -EP dormant-
@
```

BIBLIOGRAPHY

Anderson, R. H., and J. J. Gillogly, *Rand Intelligent Terminal Agent (RITA): Design Philosophy*, The Rand Corporation, R-1809-ARPA, February 1976.

Balzer, R., N. Goldman, and D. Wile, "Informality in Program Specifications," *IEEE Transactions on Software Engineering*, March 1978.

Bauer, M. A., "Programming by Examples," *Artificial Intelligence*, Vol. 12, No. 1, 1979.

Biermann, A. W., R. I. Baum, and F. E. Petry, "Speeding up the Synthesis of Programs from Traces," *IEEE Transactions on Computers*, Vol. C-24, No. 2, 1975.

Biermann, A. W., and R. Krishnawamy, "Constructing Programs from Example Computations," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, 1976.

Faught, W. S., *Applications of Exemplary Programming*, The Rand Corporation (to be published).

Girdonsky, M., and R. Neudecker, "Making the Computer Easier to Use," *IBM Research Highlights*, November 1976.

Green, C., "A Summary of the PSI Program Synthesis System," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

Heidorn, G. E., "Automatic Programming Through Natural Language Dialogue: A Survey," *Journal of Research and Development*, Vol. 20, No. 4, 1976.

Manna, Z., and R. Waldinger, "The DEDuctive ALgorithm Ur-Synthesizer," *AFIPS-NCC Conference Proceedings*, 1978.

Sacerdoti, E., "Language Access to Distributed Data with Error Recovery," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

Waterman, D. A., *Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition*, The Rand Corporation, R-2171-ARPA, February 1978.

Waterman, D. A., W. S. Faught, P. Klahr, S. J. Rosenschein, and R. Wesson, *Design Issues for Exemplary Programming*, The Rand Corporation (to be published).

RAND/R-2411-ARPA